# PARADIGM-COV

## A Multimensional Test Coverage Analysis Tool

Liliana Vilela
Departament of Informatics Engineering
Faculty of Engineering of the University of Porto
Porto, Portugal
ei08137@fe.up.pt

Ana C. R. Paiva
INESC TEC, Department of Informatics Engineering
Faculty of Engineering of the University of Porto
Porto, Portugal
apaiva@fe.up.pt

*Abstract*— **Currently, software tends to assume increasingly critical roles in our society so assuring its quality becomes ever more crucial. There are several tools and processes of software testing to help increase quality in virtually any type of software. One example is the so called Model-Based Testing (MBT) tools, that generate test cases from models. However, most of these tools have a configuration phase, where test input data is provided manually by the tester, which influences the quality of the test suite generated. By adding coverage analysis to MBT tools it is possible to give feedback and help the tester to define the configuration data needed to achieve the most valuable test suite as possible. This paper presents a tool, PARADIGM-COV, that produces coverage information both over the PARADIGM model elements (to assess if input data is adequate to cover the test goals and assess if preconditions are achievable), and during test case execution (to identify the parts of the model/code that were actually exercised).**

***Keywords-software testing, test coverage analysis, model-based testing***

## I. INTRODUCTION

As software complexity grows, it becomes increasingly difficult to assure its quality. Adding this to the increasing role of software in our daily lives, it becomes clear the ever-growing importance of software testing. Today, we have a wide range of software testing tools available that automate many aspects of the quality assurance process, which facilitates the generation of a larger quantity of test cases. Still, it is widely known that the number of tests executed does not assure system quality by itself. Rather, more than the number of tests generated, it is the quality of those tests that matters.

For one to conclude that the system has been tested thoroughly, not only must it pass the defined test cases, but one must assure that test suites themselves have met certain criteria. These criteria are usually based on metrics that allow one to know the extent to which a system is exercised by the test suites (coverage criteria), or, according to Weyuker's definition: "The purpose of testing is to uncover errors, (...) the purpose of an adequacy criterion is to access how well the testing process has been performed" [1].

Although coverage criteria and test case generation have generally been well-documented in literature, El-Far and Whittaker noted a lack of in-depth studies regarding the coverage of models in particular [2]. In fact, they are not the only authors to point coverage metrics as one of the drawbacks of Model-Based Testing (MBT). Eslamimehr has also indicated that coverage metrics prevents the MBT from being more widely applied, claiming that common metrics are insufficient, as stated previously, and that counting test cases is not enough, especially if these are automatically generated [3].

Model and code coverage information is particularly important for MBT tools, as most of them require a test configuration phase in which the tester manually provides input data that influences the quality of the test suite generated. With coverage feedback, the tester is capable of adapting the test input data to achieve the highest degree of coverage as possible, so as to improve the quality of the final test suite generated.

This work presents a tool for test coverage analysis, PARADIGM-COV, which provides diversified coverage information both over the PARADIGM graphical model elements (to assess if input data is adequate to cover the test goals and assess if preconditions are achievable), and during test case execution (to calculate the parts of the model/code that were effectively exercised).

This paper is structured into five sections. After the introduction, Section 2 describes the context of this work, followed by an overview of the developed coverage analysis tool in Section 3. Section 4 demonstrates the functionalities of the tool through a case-study. The 5th section presents the state of the art regarding coverage analysis. The last section presents conclusions and future work.

## II. STATE OF THE ART

There are not many research papers about MBT model coverage in which the coverage analysis itself is the main goal. Often times, model coverage criteria are used merely as a means to guide automatic test generation [7, 8]. Other times, instead of analyzing the model directly, the test coverage of a model is found by generating code from said model, making then a coverage analysis on that code [9]. We consider this to be code coverage at its core, and not a coverage analysis purely based on a model. Still, there exist some tools and methods that attempt to approach this problem.

The work of Weißleder is based on semantic preserving state machine transformations [10-12]. Instead of trying to achieve coverage for a complex model, he tries to achieve coverage for a simpler, but equivalent model, i.e., the stronger coverage criteria can be simulated through the use of a weaker one.

Regarding industrial practices in coverage in MBT tools, we found it somewhat hard to differentiate between the techniques used to evaluate coverage and the ones used to generate the test

cases automatically. The focus was on using coverage criteria as test generation guiding mechanisms. Such is the case at Microsoft with the Abstract State Machine Language Tool (AsmL/T), in which the definition of queries guides the test cases [7], and Spec Explorer, where coverage criteria is in practice used as test selection criteria [8, 13]. In Simulink Design Verifier tool, model coverage is used to generate additional tests, supporting coverage criteria such as Branch Coverage, test objectives or constraints [9]. Other tools that seem to fall into this set are Conformiq Qtronic and Smartesting CertifyIt [14].

In what concerns the analysis of the model itself, the most common approaches rely on exploring the state-machine derived from the model. The process, by which this is achieved, however, varies. For instance, Conformiq Qtronic tools make use of coverage checkpoints in the model [15]. With SCADE Suite, coverage analysis is done by verifying the activation of each element in the model as the system is executed. Andrade et al. use algebraic specifications, expressed in ConGu, to generate an Alloy model that obeys said specifications [16]. This technique was expanded further to support the generation of test cases for Java generic classes in particular [17].

While most commercial MBT tools found at this point do specify the basic type of functionalities provided, including coverage, they do not make it clear in what way exactly is the criterion being analyzed. Such we have found to be the case with IBM's Rational Rhapsody for instance [18]. We attribute this limited information in most commercial tools to not wanting to disclose proprietary information. A more comprehensive comparison of MBT tools in general can be found in Shafique and Labiche's report on the subject [14].

## III. User Friendly

PARADIGM-COV is developed in the context of the Pattern Based GUI Testing (PBGT) project [4]. The goal of PBGT [4,5] is to provide a means to test graphical user interfaces (GUIs) by providing generic test strategies for testing recurrent behavior. This approach relies on the fact that most GUIs end up having similar elements and behavior (the so called UI Patterns), which can be reused across several applications. PBGT defines generic test strategies able to test slightly different implementations of UI Patterns after a configuration phase. Those generic test strategies are called UI Test Patterns (UITP). A UITP defines a test strategy as a set of Test Goals (TG) with the form

$$<Goal; V; A; C; P> \quad (1)$$

where *Goal* is the name of the test strategy, *V* is a set of pairs relating the input variables with test input data, *A* is the sequence of actions to perform during test case execution, *C* is a set of checks to perform in order to evaluate the test results, and *P* is a Boolean expression defining the states in which the sequence of actions (*A*) ought to be executed.

These UITPs are defined within a graphical Domain Specific modeling Language (DSL) called PARADIGM. This DSL has structural elements (to structure the model in different levels of abstraction – *Form* element –, and to define a set of elements that can be exercised in any order – *Group* element); behavioral elements (UI Test Patterns); *Init* and *End* elements (to mark the start and end points of a model); and *connectors*, that establish relations among elements and define their sequencing.

PARADIGM-ME [4]-[6] is the modeling environment that supports building and configuring models written in PARADIGM, generates test cases from those models, establishes the mapping between model elements and GUI controls for test case execution, and executes the test cases over the application under test (AUT). PARADIGM-COV is a component of this environment.

In order to generate test cases from a PARADIGM model, the tester must go through a configuration step in which he selects the *TGs* for each UI Test Pattern within the model. For each of those *TGs*, he then provides the test data: 1) the test input data, 2) the checks to perform, and 3) the preconditions defining the states in which a *TG* may be executed. A Configured Test Goal (*TGConf*) is an instance of a *TG* in the sense that it has the test data already defined. It is possible to assign the same *TG* more than once for a UITP by providing different test data (resulting in different *TGConfs*). *TGConfs* are used in conjunction with the PARADIGM model itself for test case generation. Firstly, the tool generates test paths that guarantee full transition coverage (connectors within PARADIGM model); following, those test paths are expanded in order to exercise all *TGConfs* defined for the UITPs within the model.

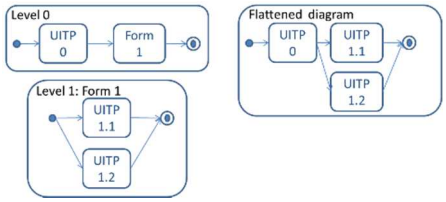


Fig 1: Diagram depicting PARADIGM-ME's path generation process.

A model can be structured into different levels of abstraction, thus, to generate test cases, PARADIGM-ME starts by flattening this model. It does this by replacing recursively all structural elements (*Forms* and *Groups*) within a hierarchical level of a model by their internal elements, present at the next level, until all *Forms* and *Groups* are discarded. After this flattening process, PARADIGM-ME calculates the set of paths (*SPaths*) that go from the *Init* to the *End* elements within the model and guarantees full connector coverage. Then, it expands every path within *SPaths* into test cases according to the *TGConf* defined for the UITP within each path. Fig. 1 illustrates a model written in two levels of abstraction (Level 0 and Level 1) on the left side and the corresponding flattened diagram on the right side. In this case *SPaths* is:

```
Path1: Init→UITP0→UITP1.1→End
Path2: Init→UITP0→UITP1.2→End
```

Let us consider that *UITP0* has two configurations defined (*c01, c02*), *UITP1.1* has one configurations (*c111*) and *UITP1.2* has two configurations (*c121, c122*), hence the final test suite would be constituted by the following test cases:

```
tc1: Init → UITP0(c01) → UITP1.1(c111) →End
tc2: Init → UITP0(c02) → UITP1.1(c111) →End
tc3: Init → UITP0(c01) → UITP1.2(c121) →End
tc4: Init → UITP0(c02) → UITP1.2(c121) →End
tc5: Init → UITP0(c01) → UITP1.2(c122) →End
tc6: Init → UITP0(c02) → UITP1.2(c122) →End
```

where *UITPx(cy)* is the instance (or *TGConf*) of the UI Test Pattern number *x* with configuration values defined in *cy*. Above, the first two test cases (*tc1*, *tc2*) are generated by expanding the first test path (*Path1*), while the remaining four (*tc3*, *tc4*, *tc5*, *tc6*) are generated by expanding the second test path (*Path2*).

We say that a test case *n* belongs to the same family of another test case *m* if they were obtained by expanding the same test path *p*. In the example given, *tc1* and *tc2* are of the same family, while *tc3*, *tc4*, *tc5*, and *tc6* belong to a second family.

## IV. Test Coverage Tool

The more complex a model gets, the harder (i.e., error prone and expensive time-wise) it is to manually keep track of all the configurations defined. One of the goals of PARADIGM-COV is to provide a diversified array of coverage information to aid the tester during the test configuration phase, in order to help him define test data that may result in the generation of test cases with the maximum level of coverage possible. As such, the PARADIGM-COV tool has three main goals: 1) **Perform model coverage analysis**: to provide feedback on whether the test configurations defined are considered 'adequate' and the preconditions are 'realizable'; 2) **Execution analysis**: to provide feedback on the model about the test execution process through dynamic coverage analysis; 3) **Code analysis**: to provide feedback on the degree to which the actual Application Under Test (AUT) code is exercised during the test execution process.

### A. Model Analysis

The first goal of PARADIGM-COV is to provide model coverage information. The model coverage analysis functionality is based on two main premises: (1) **Test Goal Analysis** – to assess whether all *TGs* within the test strategies defined for UI Test Patterns within the model were configured by the tester; (2) **Constraint Analysis** – to evaluate if the configurations provided by the tester allow reaching states where the preconditions defined hold.

**Test Goal Analysis** aims at checking if there are configurations (*TGConf*) defined for every *TG* within the test strategy of the *UITPs* within a PARADIGM model. Despite the fact that structural alone do not have test strategies associated, since their purpose is simply to allow the structuring of the model (in groups or form levels), the tool also provides goal coverage analysis on them. As such, for any structural element *Se*, its goal coverage, *gCov(Se)*, is the average of the goal coverage of the *N* elements *e* inside it:

$$gCov(Se) = \frac{\sum_{i=1}^{N} gCov(e_i)}{N} \quad (2)$$

Behavioral elements on the other hand, are *UITPs* defining a test strategy as a set of Test Goals (*TG*). So, according to Test Goal Analysis, a *UITP* is fully covered if there are configurations (*TGConf*) defined for all its *TGs* (*UITP.TG*).

$$\forall tg \in UITP.TG, \exists c \in Model.TGConf \mid c.Goal = tg \quad (3)$$

For example, the Login *UITP* defines a test strategy with two Test Goals ({*G_LVAL*, *G_LINV*}) for testing both successful and failed logins. Should the user test only successful logins, there is no guarantee the system will behave as expected when time comes to submit an invalid one. In the worst case scenario, the user could find out later that even invalid logins gave access to the system. Acceptable coverage criteria for a Login element could then be to include configurations for both valid and invalid login specifications to exercise all its Test Goals. The information relating the set of *TGs* defined by the test strategies of each *UITP* within the PARADIGM DSL can be consulted in Table 1.

The result of the Test Goal Coverage Analysis is represented on the model by painting each *UITP* element with different colors. Fully Covered: green; Uncovered (i.e., no *TGConfs* defined): red; and Partial (i.e., *TGConfs* defined for some *TGs*): yellow. With this information, the tester can easily analyze the *UITP* painted in red or yellow to complete the configuration process and allow generating higher quality test cases.

Analyzing only if there are *TGConfs* defined for every *TG* inside a *UITP* is not enough to guarantee that the corresponding test cases will be exercised. Every *TGConf* has a precondition that must hold in order for it to be executed, so it is useful to provide information about preconditions that never hold because the test input data provided by the tester does not allow it. This is the goal of **Constraint Analysis**.

Ergo, an element is considered valid constraint-wise if, for every configuration *TGConf* belonging to said element, there is a state (variables and corresponding input data) in which its associated precondition, *P*, holds (Formula 4).

$$\forall c \in \text{Model.TGConf}, \exists s\text{:State} \mid c.P(s) = \text{true} \quad (4)$$

This information is depicted as a colored box around the label of the corresponding *UITP* element. The box is green when all its constraints may hold, yellow when only some are met, and red when no preconditions are ever True. Additionally, one can see the result of Test Goal Analysis and Constraint Analysis simultaneously over the model, and get a broader picture of the coverage information.

TABLE I. Coverage criteria for each PARADIGM *UITP*

| Icon | UITP | Set of Test Goals |
|---|---|---|
|  | Call | {*G_Call*} – test the result of a call. |
|  | Find | {*G_Found*, *G_notFound*} – test searches returning and not returning values. |
|  | Input | {*G_IV*, *G_IINV*} – test for valid and invalid inputs. |
|  | Login | {*G_LV*, *G_LINV*} – test for valid and invalid authentications. |
|  | Sort | {*G_SRTASC*, *G_SRTDESC*} – test the sort for ascending and descending order. |
|  | MasterDetail | {*G_MD*} – test if exchanging the value of the master, the detail updates accordingly. |

### B. Execution Analysis

PARADIGM-COV provides information concerning the detection of discrepancies between the tests cases generated from the model and the ones effectively executed on the AUT. This dynamic analysis allows the tester to differentiate between a test that fails (i.e., the checks defined are False) from the one

that was not executed (e.g., the AUT crashed unexpectedly, or a target webpage element was not found).

While the model coverage analysis evaluates if test cases completely cover the model, execution coverage analysis evaluates if the test cases were completely executed on the AUT. So, it is possible to reach full execution coverage even when test cases do not reach full model coverage.

The execution report built by PARADIGM-COV displays several metrics pertaining to both test **check results** (*cr*) and **test execution** (*er*) coverage data. While check results indicate whether checks within a *TGConf* passed or failed, execution result simply reports whether that given *TGConf* was executed or not. The report is divided into two distinct data sections: element statistics and test case statistics.

For each *UITP* within the PARADIGM model, **element statistics** displays the percentage of *TGConfs* (*tgc*) whose checks passed, as well as the percentage of *TGConfs* that were executed. The check result coverage can thus be formalized by formula 5 where *cr* is the check result for a *tgc*: *TGConf*, and *N* is the total number of configurations (*TGConf*) within all *TGs* specified for a *UITP*.

$$\frac{\sum_{i=1}^{N}(cr(tgc) = passed)}{N} \quad (5)$$

$$\frac{\sum_{i=1}^{N}(er(tgc) = executed)}{N} \quad (6)$$

Likewise, the percentage of executed *TGConf* for each *UITP* is given by formula 6 where *er* is the execution result of the *tgc*: *TGConf*, and *N* is still the total number of configurations associated with the *TGs* of the UITP in question.

A test case is a sequence of steps, and each step is a configuration (*TGConf*) for a specific *UITP*. So, the tool presents the percentage of the overall configurations within a test case that were executed, and the percentage of the corresponding checks that passed/failed. This is **Test Case Statistics**.

Being *N* the total number of *tgc*: *TGConfs* within a test case *tc*, the percentage of passed tests within a test case is given by formula 7 and the percentage of *TGConfs* within a test case that were executed (*er*) is given by formula 8:

$$\frac{\sum_{i=1}^{N}(cr(tgc) = passed)}{N} \quad (7)$$

$$\frac{\sum_{i=1}^{N}(er(tgc) = executed)}{N} \quad (8)$$

Besides presenting overall percentages of executed and passed *TGConfs* within a test case, the tool also signals the *TGConfs* that were not executed and, additionally, for every *TGConf* executed, it reports information whether its checks passed or failed. In the latter case, it also provides a reason for failure based on exceptions that are caught by the tool during the test case execution (e.g., a UI element where an action should occur could not be found within the webpage of the AUT).

Ultimately, the aim with showing information under element and test case statistics is to ease the analysis of the test results, allowing to fix detected failures more easily.

### C. Code Coverage

When the source code of the AUT is available, it is possible to perform code coverage analysis. In simple terms, code coverage consists in tracking which parts of the code are executed while exercising the software AUT.

Indeed, while it is possible to reach full model coverage, nothing assures that full code coverage is achieved, for instance, because the model does not describe fully the AUT. Code coverage information is useful to inspect the parts of the code that were not exercised and to help to design additional test cases in order to cover them.

In what concerns the coverage criteria used, the PARADIGM-COV tool provides support for two different approaches: line coverage criteria, and block coverage criteria. Block coverage does not indicate the lines that were executed but runs faster, since the number of probe insertions is vastly inferior. In any case, the code coverage ratio for a source file is calculated using the following formula:

$$\frac{n^o\ of\ \ probes\, executed}{Total\ n^o\ of\ \ probes} \quad (9)$$

PARADIGM-COV produces a list with the files that were exercised and a report painting the files in green, yellow or red if they are fully, partially, or not exercised at all. Inside these files, code lines/blocks executed are painted in green, while the others are painted in red.

## V. Case Study

After the implementation of the PARADIGM-COV tool, we conduct a case study with the intent to: 1) showcase the basic functionalities of the tool; 2) illustrate how the tool can be used in a real-world scenario; 3) analyze the results obtained for assessing the value of tool usage. This case study is based on a real website, with source code available. The testing goals were described by a model written in PARADIGM with corresponding configurations. PARADIGM-COV was used to help the configuration activity and to assess the quality of the generated test cases.

### A. Sample Website – Museum Management Website

This website is an artwork museum management system. It allows registry new users, authenticate, insert artwork data in the database according to permission rules, manage donations, search for artworks and sort the results obtained.

### B. PARADIGM Model

A PARADIGM [19] model describes the tests to perform over the AUT. In this case, the aim is to test the registration functionality, the authentication, the search mechanism, and a subset of the administrative functions accessible only for logged-in users.

The model built for testing the Museum Management website is illustrated in figures 2. It contains one BaseMenu *Group* (meaning that its inner elements can be accessed in any order) followed by a Logout *Call*. The Registry *Call* gives

access to the Registry *Form*, detailed in another level of abstraction. The elements within the model have a graphical representation (its *UITP* icon), a label with a name, an id, and a Boolean value indicating if it is optional or mandatory element. When two elements A and B are linked by a connector it means that configurations defined for B can only be executed after those defined for A. This is the case of the connection between Login[2] and Admin[5]. Both Registry[3] *Call* and Registry[8] *Form* are optional (i.e., there will be test paths that go from *Init* directly to Login[2]), as registration is only available for non-registered users (i.e., not available for authenticated users after Login[2]). Inside Registry[8] *Form* (Fig. 3) there are several *Input UITPs* to test the input fields of the registration page for valid and invalid input data. These can be executed in any order, so they are within a *Group* element (RegistryGroup[8.1]). The RegisterButton[8.10] corresponds to the final submit button of said webpage form. To conclude, Logout[10] will test if such *Call* closes the current session and returns the website to its initial state, where the user may register or login. The configurations defined for the *UITP* within the model can be seen in Table 2. For illustration purposes we left Artworks without configurations (None).

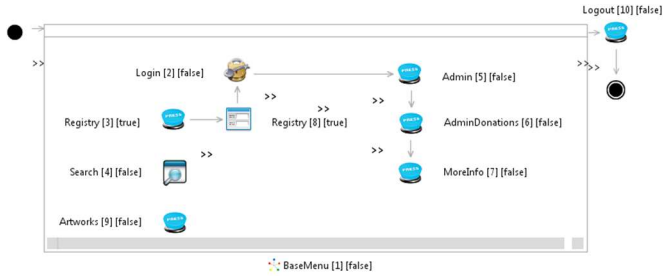


Fig 2: First level of the PARADIGM model for the artwork museum management system.

TABLE II. COMFIGURATIONS DEFINED FOR THE ELEMENTS WITHIN 1ST LEVEL MODEL

| UITP | Goals | Input Values | Checks |
|---|---|---|---|
| Call (Ids:3,5,6,7) | G_Call | None | Text present in page |
| Call (Id: 9) | None | None | None |
| Login (Id: 2) | *G_LV* | username="admin"; password ="admin" | Text present in page ="logout" |
| | *G_LINV* | username = "abc"; password = "abc" | Stay on the same page |
| Search (Id: 4) | *G_Found* | search="starry night" | Number of results=1 |

There are no preconditions defined, except for the configurations related to elements accessible only to logged-in users (Admin[5]; Donations[6]; MoreInfo[7], Logout[10]). The precondition defined for those elements is:

```
Login1_1.username==‘admin’ and
Login1_1.password==‘admin’
```

## C. Test Cases

The execution of the model above (Fig. 2) generates the following test paths:

```
{[Init;9,4,2,5,6,7,10,End],
 [Init;9,4,3,8.0,8.1,8.6,8.7,8.8,8.9,8.2,8.3,
8.4,8.5,8.10,8.11,2,5,6,7,10,End]}
```

where the elements depicted by the number 8.n, are those corresponding to the hierarchical abstraction level detailing *Form* Registry[8] not illustrated in this paper because of size restrictions.

After calculating the test paths, these are expanded according to the *TGConfs* defined for each *UITP*. For instance, considering that Login[2] has two configurations, Login_Valid and Login_Invalid (G_LV and G_LINV in Table 2) the first path would be transformed into two test cases:

```
[Init; 9, 4, 2(Login_Valid), 5, 6, 7, 10, End],
[Init; 9, 4, 2(Login_Invalid), 5, 6, 7, 10, End]
```

The same is performed by the test generation engine for the other *UITP* within such test path, in order to obtain the final executable test case.

## D. Coverage Tool Output

Fig. 3 shows the coverage information obtained by Test Goal Analysis and Constraint Analysis. Elements in red have no *TGs* configured; elements in yellow have some *TGs* defined; and elements in green have all *TGs* defined.

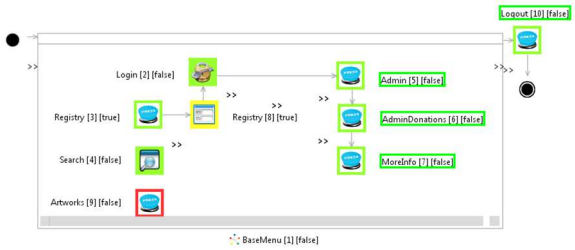


Fig 3: Result of the model coverage analysis.

The green boxes around the labels of the elements indicates that there is at least one state in which the preconditions hold. Labels without a surrounding box mean that there is no precondition defined for corresponding element.

```
Execution Report

Overall Results: 85,71% passed (92,86% executed)
        * Elem. 3               : 100% passed  (100% executed)
        * Elem. 8.2             : 100% passed  (100% executed)
        * Elem. 2               : 100% passed  (100% executed)
        * Elem. 10              : 50% passed   (50% executed)
        * Elem. 7               : 50% passed   (50% executed)
        * Elem. 6               : 100% passed  (100% executed)
        * Elem. 5               : 100% passed  (100% executed)
        * Elem. 4               : 100% passed  (100% executed)
        * Elem. 8.10            : 0% passed    (100% executed)
        * Elem. 8.7             : 100% passed  (100% executed)
        * Elem. 8.6             : 100% passed  (100% executed)
        * Elem. 8.9             : 100% passed  (100% executed)

Test Case Coverage [<testCase>: <checkResult> | <executionResult>]:
        * [0, 1, 9, 4, 2, 11]: 100% passed | 100% executed
        * [0, 1, 9, 4, 2, 5, 6, 7, 10, 11]: 100% passed | 100% executed
        * [0, 1, 9, 4, 3, 8, 8.0, 8.1, 8.6, 8.7, 8.8, 8.9, 8.2, 8.3, 8.4, 8.5
        * [0, 1, 9, 4, 3, 8, 8.0, 8.1, 8.6, 8.7, 8.8, 8.9, 8.2, 8.3, 8.4, 8.5
                                => uncovered at elements 7, 10

Test Trace [<executionOrder>) <elementID>: <testResult>]:
        (executed tests based on generated script)
           1) 4                 : passed
           2) 2                 : passed
           3) 4                 : passed
           4) 2                 : passed
```

Fig 4: Part of the execution report with check success and test case execution measurements.

To illustrate the execution analysis, we forced a crash near the end of the test case execution process, as to showcase non-

executed tests. Once the testing finishes, a tabbed view shows the results (Fig. 4). As shown, the report informs that 85.71% of all *TGConfs* passed and 92.86% of all *TGConfs* were executed.

By analyzing the reports produced, the tester needs to understand why elements 10 and 8.10 did not reach 100% passed and executed. For element 10, it happens because we forced a crash. For element 8.10, all its configurations (*TGConf*) were actually executed, but none has passed (i.e., checks return False). So, either the AUT or the model is wrong. The tester must analyze the checks in the *TGConfs* defined for this element and decide if it is a bug within the AUT or a problem with the specification. This analysis is facilitated by the expanded Test Case Statistics, dubbed 'test trace' in Fig. 4, which displays the input data associated with failed or non-executed *TGConfs*.

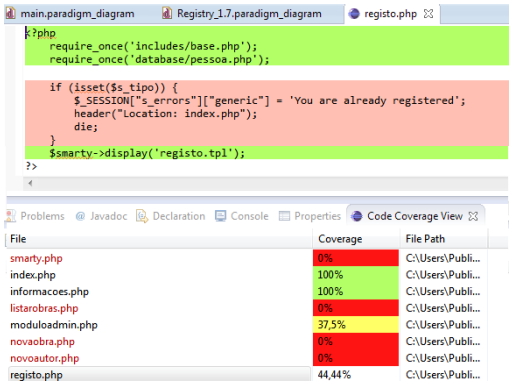


Fig 5: Code coverage for a single file and overall results for the list of analyzed files.

The result of code coverage analysis can be seen in Fig. 5. Element 3 is fully covered on the model, but does not reach full code coverage (as shown by the code block signaled in red). The uncovered code block pertains to a situation occurring when logged-in users attempt to access this page. Such can happen if an authenticated user tries to access this functionality directly by typing the URL. The model built does not aim to test this functionality, so the test cases generated do not exercise it. However, by analyzing the code coverage results, the tester can build additional tests to execute and test this block of code.

## VI. Conclusions

This paper presents a Coverage Analysis tool (PARADIGM-COV) that produces model and code coverage information in the context of a MBT project called PBGT. The main advantage of the PARADIGM-COV tool is its capability to perform a broad coverage analysis over different aspects, either presenting information about test requirements coverage (Test Goal Analysis), model coverage (Restriction Analysis – to check if preconditions ever hold), test case coverage (passed/executed metrics through Execution Analysis), as well as code coverage.

Through the case study, it was possible to conclude that there is not one superior metric to evaluate the quality of test cases, but rather, the best approach is the combination of metrics to merge their strengths and obtain broader coverage information.

In the future, we intend to implement code coverage analysis for other programming languages besides PHP and improve the usability of PARADIGM-COV tool.

## Acknowledgment

This work is financed by the ERDF – European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020554.

## References

[1] Weyuker, E.J.: Axiomatizing software test data adequacy. IEEE Transactions on Software Engineering. SE-12, 1128 –1138 (1986).

[2] El-Far, I.K., Whittaker, J.A.: Model-Based Software Testing. Encyclopedia of Software En-gineering. John Wiley & Sons, Inc. (2002).

[3] Eslamimehr, M.M.: The Survey of Model Based Testing and Industrial Tools, (2008).

[4] Monteiro, T., Paiva, A.C.R.: Pattern Based GUI Testing Modeling Environment. 4th International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software (TESTBEDS). (2013).

[5] Moreira, R., Paiva, A. C. R., Memon, A.: A Pattern-Based Approach for GUI Modelling and Testing. The 24th IEEE International Symposium on Software Reliability Engineering (ISSRE), 2013,

[6] Cunha, M., Paiva, A.C.R., Ferreira, H.S., Abreu, R.: PETTool: A pattern-based GUI testing tool. Software Technology and Engineering (ICSTE), 2010 2nd International Conference on. pp. V1–202 (2010).

[7] Stobie, K.: Model Based Testing in Practice at Microsoft. Electronic Notes in Theoretical Computer Science. 111, 5–12 (2005).

[8] Grieskamp, W., Kicillof, N., Stobie, K., Braberman, V.: Model-based quality assurance of protocol documentation: tools and methodology. Software Testing, Verification and Relia-bility. 21, 55–71 (2011).

[9] Pretschner, A.: Model-based testing. 27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings. pp. 722 – 723 (2005).

[10] Weißleder, S.: Simulated Satisfaction of Coverage Criteria on UML State Machines. 2010 Third International Conference on Software Testing, Verification and Validation (ICST). pp. 117 –126 (2010).

[11] Weißleder, S.: Test models and coverage criteria for automatic model-based test generation with UML state machines, (2010).

[12] Weißleder, S: Coverage Simulator, http://covsim.sourceforge.net/.

[13] Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In: Hierons, R.M., Bowen, J.P., and Harman, M. (eds.) Formal Methods and Testing. pp. 39–76. Springer Berlin Heidelberg (2008).

[14] Shafique, M., Labiche, Y.: A systematic review of model based testing tool support. Technical Report SCE-10-04, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada (2010).

[15] Huima, A.: Implementing Conformiq Qtronic. In: Petrenko, A., Veanes, M., Tretmans, J., and Grieskamp, W. (eds.) Testing of Software and Communicating Systems. pp. 1–12. Springer Berlin Heidelberg (2007).

[16] Andrade, F.R., Faria, J.P., Paiva, A.C.R.: Test generation from bounded algebraic specifications using alloy. ICSOFT (2). 192–200 (2011).

[17] Andrade, F.R., Faria, J.P., Lopes, A., Paiva, A.C.R.: Specification-Driven Unit Test Genera-tion for Java Generic Classes. In: Derrick, J., Gnesi, S., Latella, D., and Treharne, H. (eds.) Integrated Formal Methods. pp. 296–311. Springer Berlin Heidelberg (2012).

[18] IBM: Safety-related software development using a model-based testing workflow, http://www.ibm.com/developerworks/rational/library/safety-related-software-development/index.html.

[19] Moreira, Rodrigo M. L. M and Paiva, Ana C. R., A GUI Modeling DSL for Pattern-Based GUI Testing – PARADIGM, in Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), Maciaszek, Leszek A. and Filipe, Joaquim (ed.), SciTePress (2014).